

2

REPORT DOCUMENTATION PAGE

Form Approved

OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden, to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE

June 28, 1995

3. REPORT TYPE AND DATES COVERED

Final

4. TITLE AND SUBTITLE:

Ada Compiler Validation Summary Report, VC# 950622I1.11391
Tartan, Inc. -- Tartan Ada VMS/1750A, Version 5.1

5. FUNDING NUMBERS



6. AUTHOR(S)

IABG, Abt. ITE

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Ada Validation Facility
IABG, Abt. ITE
Einsteinstrasse 20
D-85521 Ottobrunn, GERMANY

8. PERFORMING ORGANIZATION
REPORT NUMBER

IABG-VSR 116

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office, Defense Information System Agency
Code JEXEV, 701 S. Courthouse Rd., Arlington, VA
22204-2199

10. SPONSORING/MONITORING
AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

19950905 012

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 22 June 1995.
Host Computer System: VAXstation 4000/60 under VMS, Version 5.5
Target Computer System: Fairchild F9450 on an SBC-50 Board (MIL-STD-1750A, bare machine)

14. SUBJECT TERMS

Ada Programming Language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

63

16. PRICE

17. SECURITY CLASSIFICATION
OF REPORT

UNCLASSIFIED

18. SECURITY CLASSIFICATION
OF THIS PAGE

UNCLASSIFIED

19. SECURITY CLASSIFICATION
OF ABSTRACT

UNCLASSIFIED

20. LIMITATION OF ABSTRACT

UNCLASSIFIED

NSN 7540-01-280-5500

ALSO QUALITY INSPECTED

AVF Control Number: IABG-VSR 116
28 June, 1995

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 950622I1.11391
Tartan, Inc.
Tartan Ada VMS/1750A Version 5.1
VAXstation 4000/60 =>
Fairchild F9450 on an SBC-50 Board

Prepared By:
IABG, Abt. ITE
Einsteinstr. 20
D-85521 Ottobrunn
Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 22 June, 1995.

Compiler Name and Version: Tartan Ada VMS/1750A Version 5.1

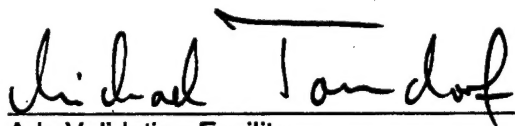
Host Computer System: VAXstation 4000/60 under VMS Version 5.5

Target Computer System: Fairchild F9450 on an SBC-50 Board (MIL-STD-1750A, bare machine)


See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 95062211.11391 is awarded to Tartan, Inc. This certificate expires on 31 March 1998.

This report has been reviewed and is approved.



Ada Validation Facility
IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
D-85521 Ottobrunn
Germany



Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311, USA



Ada Joint Program Office
Center for Information Management
Defense Information Systems Agency
Don Reifer, Director
Arlington VA 22204, USA

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution	
Availability Codes	
Dist	Avail and/or Special
A-1	

Declaration of Conformance

Customer: Tartan, Inc.

Certificate Awardee: Tartan, Inc.

Ada Validation Facility: IABG

ACVC Version: 1.11

Ada Implementation:

Ada CompilerName and Version: Tartan Ada VMS/1750A Version 5.1

Host Computer System: VAXstation 4000/60 under VMS V5.5

Target Computer System: Fairchild F9450 on an SBC-50 board
(MIL-STD-1750A) (bare machine)

Declaration:

I, the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A, ISO 8652-1987, FIPS 119 as tested in this validation and documented in the Validation Summary Report.



Steve McMahon
Senior Vice President

1/28/95

Date

TABLE OF CONTENTS

CHAPTER 1		
1.1	USE OF THIS VALIDATION SUMMARY REPORT	1- 1
1.2	REFERENCES	1- 2
1.3	ACVC TEST CLASSES	1- 2
1.4	DEFINITION OF TERMS	1- 3
CHAPTER 2		
2.1	WITHDRAWN TESTS	2- 1
2.2	INAPPLICABLE TESTS	2- 1
2.3	TEST MODIFICATIONS	2- 4
CHAPTER 3		
3.1	TESTING ENVIRONMENT	3- 1
3.2	SUMMARY OF TEST RESULTS	3- 2
3.3	TEST EXECUTION	3- 2
APPENDIX A MACRO PARAMETERS		
APPENDIX B COMPILATION SYSTEM OPTIONS		
APPENDIX C APPENDIX F OF THE Ada STANDARD		

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro92] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro92]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161, USA

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772, USA

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language,
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

- [Pro92] Ada Compiler Validation Procedures,
Version 3.1, Ada Joint Program Office, August 1992.

- [UG90] Ada Compiler Validation Capability User's Guide,
3 April 1990.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

INTRODUCTION

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1), and possibly removing some inapplicable tests (see section 2.2 and [UG90]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organisation	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

INTRODUCTION

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realised or is attainable on the Ada implementation for which validation status is realised.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organisation for Standardisation.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro92].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.

INTRODUCTION

Withdrawn
test

A test found to be incorrect and not used in conformity testing.
A test may be incorrect because it has an invalid test objective,
fails to meet its test objective, or contains erroneous or illegal
use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following 104 tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is November 22, 1993.

B27005A	E28005C	B28006C	C32303A	C34006D	C35507K
C35507L	C35507N	C35507O	C35507P	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	C35310A	B41308B
C43004A	C45114A	C45346A	C45612A	C45612B	C45612C
C45651A	C46022A	B49008A	B49008B	A54B02A	C55B06A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
C83026A	B83026B	C83041A	B85001L	C86001F	C94021A
C97116A	C98003B	BA2011A	CB7001A	CB7001B	CB7004A
CC1223A	BC1226A	CC1226B	BC3009B	BD1B02B	BD1B06A
BD1B08A	BD2A02A	CD2A21E	CD2A23E	CD2A32A	CD2A41A
CD2A41E	CD2A87A	CD2B15C	BD3006A	BD4008A	CD4022A
CD4022D	CD4024B	CD4024C	CD4024D	CD4031A	CD4051D
CD5111A	CD7004C	ED7005D	CD7005E	AD7006A	CD7006E
AD7201A	AD7201E	CD7204B	AD7206A	BD8002A	BD8004C
CD9005A	CD9005B	CDA201E	CE2107I	CE2117A	CE2117B
CE2119B	CE2205B	CE2405A	CE3111C	CE3116A	CE3118A
CE3411B	CE3412B	CE3607B	CE3607C	CE3607D	CE3812A
CE3814A	CE3902B				

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

IMPLEMENTATION DEPENDENCIES

The following 285 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

The following 21 tests check for the predefined type `SHORT_INTEGER`; for this implementation, there is no such type:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C45504E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than `INTEGER`, `LONG_INTEGER`, or `SHORT_INTEGER`; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`; for this implementation, there is no such type.

A35801E checks that `FLOAT'FIRST..FLOAT'LAST` may be used as a range constraint in a floating-point type declaration; for this implementation, that range exceeds the range of safe numbers of the largest predefined floating-point type and must be rejected. (See section 2.3.)

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater; for this implementation, `MAX_MANTISSA` is less than 47.

C45536A, C46013B, C46031B, C46033B, and C46034B contain length clauses that specify values for `'SMALL` that are not powers of two or ten; this implementation does not support such values for `'SMALL`.

C45624A..B (2 tests) check that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, `MACHINE_OVERFLOW` is `TRUE`.

IMPLEMENTATION DEPENDENCIES

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C instantiate generic units before their bodies are compiled; this implementation creates a dependence on generic units as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (see 2.3.)

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

The following 264 tests check operations on sequential, text, and direct access files; this implementation does not support external files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)

IMPLEMENTATION DEPENDENCIES

CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B, and CE3107A use an illegal file name in an attempt to create a file and expect NAME_ERROR to be raised; this implementation does not support external files and so raises USE_ERROR. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see Section 1.3) were required for 106 tests.

The following 81 tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24007A	B24009A	B25002B	B32201A	B33204A
B33205A	B35701A	B36171A	B36201A	B37101A	B37102A
B37201A	B37202A	B37203A	B37302A	B38003A	B38003B
B38008A	B38008B	B38009A	B38009B	B38103A	B38103B
B38103C	B38103D	B38103E	B43202C	B44002A	B48002A
B48002B	B48002D	B48002E	B48002G	B48003E	B49003A
B49005A	B49006A	B49006B	B49007A	B49007B	B49009A
B4A010C	B54A20A	B54A25A	B58002A	B58002B	B59001A
B59001C	B59001I	B62006C	B67001A	B67001B	B67001C
B67001D	B74103E	B74104A	B74307B	B83E01A	B85007C
B85008G	B85008H	B91004A	B91005A	B95003A	B95007B
B95031A	B95074E	BA1001A	BC1002A	BC1109A	BC1109C
BC1206A	BC2001E	BC3005B	BD2A06A	BD2B03A	BD2D03A
BD4003A	BD4006A	BD8003A			

E28002B was graded passed by Evaluation and Test Modification as directed by the AVO. This test checks that pragmas may have unresolvable arguments, and it includes a check that pragma LIST has the required effect; but, for this implementation, pragma LIST has no effect if the compilation results in errors or warnings, which is the case when the test is processed without modification. This test was also processed with the pragmas at lines 46, 58, 70 and 71 commented out so that pragma LIST had effect.

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO. The compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. LRM 3.5.7:12).

Tests C45524A..E (5 tests) were graded passed by Test Modification as directed by the AVO. These tests expect that a repeated division will result in zero; but the Ada standard only requires that the result lie in the smallest safe interval. Thus, the tests were modified to check that the result was within the smallest safe interval by adding the following code after line 141; the modified tests were passed:

IMPLEMENTATION DEPENDENCIES

```
ELSIF VAL <= F'SAFE_SMALL  
  THEN COMMENT ("UNDERFLOW SEEMS GRADUAL");
```

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package report's body, and thus the packages' calls to function Report.Ident_Int at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

B83E01B was graded passed by Evaluation Modification as directed by the AVO. This test checks that a generic subprogram's formal parameter names (i.e. both generic and subprogram formal parameter names) must be distinct; the duplicated names within the generic declarations are marked as errors, whereas their recurrences in the subprogram bodies are marked as "optional" errors--except for the case at line 122, which is marked as an error. This implementation does not additionally flag the errors in the bodies and thus the expected error at line 122 is not flagged. The AVO ruled that the implementation's behavior was acceptable and that the test need not be split (such a split would simply duplicate the case in B83E01A at line 15).

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C were graded inapplicable by Evaluation Modification as directed by the AVO. These tests instantiate generic units before those units' bodies are compiled; this implementation creates dependences as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete, and the objectives of these tests cannot be met.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 & AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by compiling the separate files in the following order (to allow re-compilation of obsolete units), and all intended errors were then detected by the compiler:

BC3204C: C0, C1, C2, C3M, C4, C5, C6, C3M

BC3205D: D0, D1M, D2, D1M

BC3204D and BC3205C were graded passed by Test Modification as directed by the AVO. These tests are similar to BC3204C and BC3205D above, except that all compilation units are contained in a single compilation. For these two tests, a copy of the main procedure (which later units make obsolete) was appended to the tests; all expected errors were then detected.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-ten value as small for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal smalls may be omitted.

IMPLEMENTATION DEPENDENCIES

AD9001B and AD9004A were graded passed by Processing Modification as directed by the AVO. These tests check that various subprograms may be interfaced to external routines (and hence have no Ada bodies). This implementation requires that a file specification exists for the foreign subprogram bodies. The following command was issued to the Librarian to inform it that the foreign bodies will be supplied at link time (as the bodies are not actually needed by the program, this command alone is sufficient):

```
interface -sys -L=library ad9001b & ad9004a
```

CE2103A, CE2103B, and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when `USE_ERROR` is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical information about this Ada implementation, contact:

Mr Wayne Lieberman
Ada Business Unit Leader
Tartan Inc.
300 Oxford Drive
Monroeville, PA 15146, USA
Tel. (412) 856-3600

For sales information about this Ada implementation, contact:

Mr Keith Franz
Vice President Sales
Tartan Inc.
300 Oxford Drive
Monroeville, PA 15146, USA
Tel. (412) 856-3600

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

PROCESSING INFORMATION

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro92].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

Summary of Test Counts		
a	Total Number of Applicable Tests	3452
b	Total Number of Withdrawn Tests	104
c	Processed Inapplicable Tests	65
d	Non-Processed I/O Tests	264
e	Non Processsd Floating Point Precision Tests	285
f	Total Number of Inapplicable Tests (c+d+e)	614
g	Total Number of Tests for ACVC 1.11 (a+b+f)	4170

3.3 TEST EXECUTION

A magnetic data cartridge containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic data cartridge were loaded on a computer with an attached tape drive and copied directly onto the host computer using networking facilities.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the communications link, an ethernet interface, and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were for compiling:

PROCESSING INFORMATION

<code>/replace</code>	forces the compiler to accept an attempt to compile a unit imported from another library which is normally prohibited.
<code>/nosave_source</code>	suppresses the creation of a registered copy of the source code in the library directory for use by the REMAKE and MAKE subcommands.
<code>/list=always</code>	forces a listing to be produced, default is to only produce a listing when an error occurs

For this validation the default optimization level -Op2 was used. No explicit Linker options were set.

Test output, compiler and linker listings, and job logs were captured on magnetic data cartridge and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG90]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for \$MAX_IN_LEN -- also listed here. These values are expressed here as Ada string aggregates, where

Macro Parameter	Macro Value
\$MAX_IN_LEN	240
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	"" & (1..V/2 => 'A') & ""
\$BIG_STRING2	"" & (1..V-1-V/2 => 'A') & '1' & ""
\$BLANKS	(1..V-20 => ' ')
\$ILLEGAL_EXTERNAL_FILE_NAME1	"ILLEGAL_EXTERNAL_FILE_NAME1" & (1..V => '-')
\$ILLEGAL_EXTERNAL_FILE_NAME2	"ILLEGAL_EXTERNAL_FILE_NAME2" & (1..V => '-')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	"" & (1..V-2 => 'A') & ""

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
\$ACC_SIZE	16
\$ALIGNMENT	1
\$COUNT_LAST	32766
\$DEFAULT_MEM_SIZE	65536
\$DEFAULT_STOR_UNIT	16
\$DEFAULT_SYS_NAME	MIL_STD_1750A
\$DELTA_DOC	2#1.0#E-31
\$ENTRY_ADDRESS	SYSTEM.ADDRESS'(16#0000_000D#)
\$ENTRY_ADDRESS1	SYSTEM.ADDRESS'(16#0000_000E#)
\$ENTRY_ADDRESS2	SYSTEM.ADDRESS'(16#0000_00=f#)
\$FIELD_LAST	20
\$FILE_TERMINATOR	' '
\$FIXED_NAME	NO_SUCH_TYPE
\$FLOAT_NAME	NO_SUCH_TYPE
\$FORM_STRING	""
\$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
\$GREATER_THAN_DURATION	100_000.0
\$GREATER_THAN_DURATION_BASE_LAST	100_000_000.0
\$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
\$GREATER_THAN_FLOAT_SAFE_LARGE	1.7014111E+38

MACRO PARAMETERS

\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.7014111E+38
\$HIGH_PRIORITY	200
\$INAPPROPRIATE_LINE_LENGTH	-1
\$INAPPROPRIATE_PAGE_LENGTH	-1
\$INCLUDE_PRAGMA	PRAGMA INCLUDE ("A28006D1.TST")
\$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006F1.TST")
\$INTEGER_FIRST	-32768
\$INTEGER_LAST	32767
\$INTEGER_LAST_PLUS_1	32768
\$INTERFACE_LANGUAGE	C
\$LESS_THAN_DURATION	-100_000.0
\$LESS_THAN_DURATION_BASE_FIRST	-131_073.0
\$LINE_TERMINATOR	' '
\$LOW_PRIORITY	10
\$MACHINE_CODE_STATEMENT	Two_Opnds'(LR,(R_am,R0),(R_am,R1));
\$MACHINE_CODE_TYPE	Instruction_Mnemonic
\$MANTISSA_DOC	31
\$MAX_DIGITS	9
\$MAX_INT	2_147_483_647
\$MAX_INT_PLUS_1	2_147_483_648
\$MIN_INT	-2_147_483_648
\$NAME	NO_SUCH_TYPE
\$NAME_LIST	MIL_STD_1750A
\$NEG_BASED_INT	16#FFFFFFFFE#

MACRO PARAMETERS

\$NEW_MEM_SIZE	1048576
\$NEW_STOR_UNIT	16
\$NEW_SYS_NAME	MIL_STD_1750A
\$PAGE_TERMINATOR	' '
\$RECORD_DEFINITION	record Operation: Instruction_Mnemonic; Operand_1: Operand; Operand_2: Operand; end record;
\$RECORD_NAME	Two_Opnds
\$TASK_SIZE	16
\$TASK_STORAGE_SIZE	1024
\$TICK	0.0001
\$VARIABLE_ADDRESS	SYSTEM.ADDRESS'(16#0000_0004#)
\$VARIABLE_ADDRESS1	SYSTEM.ADDRESS'(16#0000_0005#)
\$VARIABLE_ADDRESS2	SYSTEM.ADDRESS'(16#0000_0006#)

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

The following VMS command line options may be specified:

1750a	Invokes the 1750A-targeted cross compiler.
-------	--------------------------------------------

9x		
no9x (Default)		Notifies the compiler that the source file to be compiled contains Ada 9X constructs.

body dependencies[=*argument*]

Controls whether an optimization file which contains special optimization information is produced when the unit being compiled is a body. Also, controls the use of optimization information available from units named in the context clause of the current unit.

The following *arguments* can be specified:

all The compiler will produce an optimization (.opt) file, when the unit being compiled is a body. When another unit is compiled which refers to this unit in its context clause, a dependency may be created on this unit's body (in addition to the specification) due to the utilization of this optimization information. The compiler will read the optimization files of units in the current unit's compilation closure to obtain information which may be used to improve the optimizations performed on the current unit. Dependencies will be created on both the specification and the body (if any) of the units from which optimization information is utilized. This option will allow maximum optimization at the expense of increased recompilations when changes are made.

`import_only` The compiler will *not* produce an optimization file when the unit being compiled is a body (effectively preventing the creation of dependencies on this body). The compiler will read the optimization files of units in the current unit's compilation closure to obtain information which may be used to improve the optimizations performed on the current unit. Dependencies will be created on units from which optimization information is utilized.

minimal The compiler will neither produce an optimization file when the unit being compiled is a body (effectively preventing the creation of dependencies on this unit), nor will the compiler attempt to read optimization information from units in the current unit's compilation closure (preventing the creation of a dependency on other body units). When compiling an entire system, this strategy will lead to minimal dependencies between the compilation units in the system.

The default condition is `body_dependencies=import_only`.

`cross_reference`
`nocross_reference (Default)` Controls whether the compiler generates cross-reference information in the object code file (see *AdaRef Manual*).

`debug`
`nodebug (Default)` Produces debugging information for AdaScope, the Tartan Ada symbolic debugger. It is not necessary for all object modules to include debugging information to obtain a linkable image, but use of this option is encouraged for all compilations. No execution-time penalty is incurred with this option.

`enumeration_images (Default)`
`noenumeration_images` Causes the compiler to omit data segments with the text of enumeration literals. This text is normally produced for exported enumeration types to support the text attributes ('IMAGE, 'VALUE and 'WIDTH). You should use `noenumeration_images` only when you can guarantee that no unit that will import the enumeration type will use any of its text attributes. However, if you are compiling a unit with an enumeration type that is not visible to other compilation units, this option is not needed. The compiler can recognize when the text attributes are not used and will not generate the supporting strings. This option is intended to reduce the size of execution images for embedded systems.

`error_limit=n` Stops compilation and produces a listing after *n* errors are encountered, where *n* is an integer in the range 1..255. The default value for *n* is 255.

`fixup[=argument]` When package `MACHINE_CODE` is used, controls whether the compiler attempts to alter operand address modes when those address modes are used incorrectly.

The following *arguments* can be specified:

<code>quiet</code>	The compiler attempts to generate extra instructions to fix incorrect address modes in the array aggregates operand field.
<code>warn</code>	The compiler attempts to generate extra instructions to fix incorrect address modes. A warning message is issued if such a correction is required.
<code>none</code>	The compiler does <i>not</i> attempt to fix any machine code insertion that has incorrect address modes. An error message is issued for any machine code insertion that is incorrect.

When no form of this option is supplied in the command line, the default condition is `fixup=quiet`. For more information on machine code insertions, refer to section 4.10.

`length=n` Specifies the number of lines printed on a page of the listing file, where *n* is an integer in the range 6..9999. The default value for *n* is 60.

`library=[project:]library` Selects the library and optionally the project for this compilation. This option takes effect after all commands from the librarian initialization

`lis_file=file`

`file`⁷ have been executed, thereby possibly overriding its effects.

The argument *file* specifies the file name to be used for a listing file if one is produced.

`list [=argument]`
`nolist`

Controls whether a listing file is produced. If produced, the file has the source file name and a `.lis` extension unless the `lis_file` option is also specified.

The following *arguments* can be specified:

<code>always</code>	Always produces a listing file
<code>never</code>	Never produces a listing file, equivalent to <code>nolist</code>
<code>error</code>	Produces a listing file only if a diagnostic message is issued

When no form of this option is supplied in the command line, the default condition is `list=error`. When the `list` option is supplied without an argument, the default argument is `always`.

`long_address_arithmetic` (Default)
`nolong_address_arithmetic`

Instructs the compiler to use 16-bit arithmetic rather than 32-bit arithmetic for certain addressing expressions. This leads to smaller code size.

Certain address calculations involving array subscripts may require 32-bit arithmetic. This typically happens when indexing within arrays that are allocated near to the `16#7FFF#..16#F000#` address boundary. By using this option, the programmer asserts that no such problems will arise in the current compilation unit.

The compiler automatically uses 16-bit arithmetic anywhere it is guaranteed to be safe to do so. This option affects only those calculations that might require 32-bit arithmetic.

`machine_code [=argument]`

Controls whether the compiler produces an assembly code file in addition to an object code file. If produced, the file has the compilation unit name and a `.asm` or `.sasm` extension (sec. 3.5). The assembly code file is not intended to be input to an assembler, but serves as documentation only.

The following *arguments* can be specified:

<code>none</code>	Do not produce an assembly code file.
<code>interleave</code>	Produces an assembly code file which interleaves source code with the machine code. Ada source appears as assembly language comments.
<code>nointerleave</code>	Produces an assembly code file without interleaving.

When no form of this option is supplied in the command line, the default condition is `machine_code=none`. When the `machine_code` option is supplied without an argument, the default argument is `nointerleave`.

⁷VMS: adalib.ini

`messages [=argument]`

Controls the type of messages that will be generated by the compiler.

The following *arguments* can be specified:

<code>errors_only</code>	Reports only errors.
<code>warnings</code>	Reports errors and warnings.
<code>all</code>	Reports errors, warnings, and informational messages.

The default condition is `messages=warnings`.

`optimize=argument`

Controls the level of optimization performed by the compiler. However, when the code being compiled contains an `OPTIMIZE` pragma, the pragma takes precedence over the specification of this command line option.

The following optimization levels can be specified:

<code>minimum</code>	Performs context determination, constant folding, algebraic manipulation, and short circuit analysis. Pragma <code>INLINEs</code> are <i>not</i> obeyed.
<code>low</code>	Performs <i>minimum</i> optimizations plus evaluation order determination as well as common subexpression elimination and equivalence propagation within basic blocks. Again, pragma <code>INLINEs</code> are <i>not</i> obeyed.
<code>standard</code>	(default) Best tradeoff for space/time. Performs <i>low</i> optimizations plus flow analysis for common subexpression elimination and equivalence propagation across basic blocks, invariant hoisting, dead code elimination, assignment killing, strength reduction, lifetime analysis for improved register allocation, tail recursion elimination, interprocedural side-effect analysis and some inline-expansion.
<code>time</code>	Performs <i>standard</i> optimizations plus loop unrolling and aggressive inline expansion. This optimization level usually produces the fastest code, however, optimization level <i>standard</i> may produce faster code under certain circumstances.
<code>space</code>	Performs <i>standard</i> optimizations minus any optimization that may increase code size. This optimization level usually produces the smallest code, however, optimization level <i>standard</i> may produce smaller code under certain circumstances.

`parse`
`noparse (Default)`

Loads a syntactically correct compilation unit(s) in the source file into the program library as a parsed unit(s). Parsed units are, by definition, inconsistent. This option allows users to load units into the library without regard to correct compilation order. The librarian's *remake*

	subcommand ⁸ is subsequently used to compile the units in the correct sequence (sec. 10.2.5.2).
phases nophases (Default)	Controls whether the compiler announces each phase of processing as it occurs. These phases indicate the progress of the compilation.
profile nopprofile (Default)	Produces the additional code and data necessary to perform profile analysis (see <i>AdaTrak Manual</i>).
refine norefine (Default)	Controls whether the compiler, when compiling a library unit, determines whether the unit is a refinement of its previous version and, if so, does not make dependent units obsolete. A warning message is given if the unit is not a refinement of its previous version. The <i>no update</i> option ⁹ can be used in conjunction with this option to check for possible refinements without risking a change to the program library.
remember noremember (Default)	Causes the compiler options specified for this compilation unit to be saved in the program library. These options will be used when a subsequent <i>(re)make</i> command is issued on this unit, unless overridden by compiler options specified on the <i>(re)make</i> command line (sec. 10.2.5).
replace noreplace (Default)	Forces the compiler to accept an attempt to compile a unit imported from another library which is normally prohibited.
save_source (Default) nosave_source	Controls the creation of a registered copy of the source code in the library directory for use by the librarian <i>remake</i> subcommand. If this option is specified, it is very important to provide AdaScope with the directories in which the source files reside. See the section "Locating Source Files" in the <i>AdaScope Manual</i> .
suppress [= (argument, ...)]	Suppresses the specific checks identified by the arguments supplied. The parentheses can be omitted if only one argument is supplied. Invoking this option will <i>not</i> remove all checks if the resulting code without checks will be less efficient. The <i>suppress</i> option has the same effect as a global pragma <i>SUPPRESS</i> applied to the source file. If the source program also contains a pragma <i>SUPPRESS</i> , a given check is suppressed if either the pragma or the option specifies it; i.e., the effect of a pragma <i>SUPPRESS</i> cannot be negated with the command line option. The <i>suppress</i> option cannot be negated. The following <i>arguments</i> can be specified: all Suppress all checks. This argument is the default when no argument is supplied.

⁸VMS: remake

⁹VMS: /nouupdate

ACCESS_CHECK	As specified in the Ada LRM, Section 11.7.
CONSTRAINT_CHECK	Equivalent to specifying all of the following: ACCESS_CHECK, INDEX_CHECK, DISCRIMINANT_CHECK, LENGTH_CHECK, RANGE_CHECK.
DISCRIMINANT_CHECK	As specified in the Ada LRM, Section 11.7.
DIVISION_CHECK	Suppresses compile-time checks for division by zero, but the hardware does not permit efficient runtime checks, so none are done.
ELABORATION_CHECK	As specified in the Ada LRM, Section 11.7.
INDEX_CHECK	As specified in the Ada LRM, Section 11.7.
LENGTH_CHECK	As specified in the Ada LRM, Section 11.7.
none	All checks are performed, no checks are suppressed. This condition is the default if the option is not supplied on the command line.
OVERFLOW_CHECK	Suppresses compile-time checks for overflow, but the hardware does not permit efficient runtime checks, so none are done.
RANGE_CHECK	As specified in the Ada LRM, Section 11.7.
STORAGE_CHECK	As specified in the Ada LRM, Section 11.7. Suppresses only stack checks in generated code, not the checks made by the allocator as a result of a new operation.
syntax_only nosyntax_only (Default)	Examines units for syntax errors, then stops compilation. No semantic checking is performed. Nothing is entered in the program library. <i>No library need be specified when using this option.</i>
target= <i>argument</i>	This option provides access to proprietary features for several 1750A design variations. The following <i>arguments</i> can be specified:
	HONEYWELL_GVSC_RH_1750A Aligns data objects 32-bits or larger in memory so that each begins on an even numbered address. The compiler will not use the MOV instruction when it can determine during compilation that the source object will overlap the destination object.
	MIL_STD_1750A Default argument. No special features accessed.
	TI_SLC1750 The compiler will not use the MOV instruction when it can determine during compilation that the source object will overlap the destination object.

COMPILATION SYSTEM MANUAL

WESTINGHOUSE_VAMP Causes the compiler to emit two single store instructions instead of a single double store instruction.

update (Default)
nouupdate

Controls whether the program library will be updated with the result of this compilation.

width=*n*

Specifies the line width used in a listing, where *n* is an integer in the range 80..132. The default value for *n* is 80.¹⁰

¹⁰Note that 10 fewer characters than the user specifies on the command line will actually appear on a line in the listing file due to the left and right margins and the line numbers.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, are outlined below for convenience.

package STANDARD is

...

type INTEGER is range -32768 .. 32767;

type FLOAT is digits 6 range -16#0.8000_00#E+32 .. 16#0.7FFF_FF#E+32;

type LONG_INTEGER is range -2147483648 .. 2147483647;

type LONG_FLOAT is digits 9 range -16#0.8000_0000_00#E+32 ..
16#0.7FFF_FFFF_FF#E+32 ;

type DURATION is delta 0.0001 range -86400.0 .. 86400.0;

-- Duration'Small = 2#1.0#E-14

...

end STANDARD;

APPENDIX F OF THE Ada STANDARD

APPENDIX F OF MIL-STD-1815A

This chapter contains the required Appendix F to the LRM, which is Military Standard, Ada Programming Language, ANSI/MIL-STD-1815A.

4.1. PRAGMAS

4.1.1. Predefined Pragmas

The Tartan Ada Compiler supports all of the predefined pragmas described in the LRM, Annex B.

- pragma CONTROLLED (sec. 4.1.1.1)
- pragma ELABORATE
- pragma INLINE (sec. 4.1.1.2)
- pragma INTERFACE (sec. 4.1.1.3)
- pragma LIST
- pragma MEMORY_SIZE (sec. 4.1.1.4)
- pragma OPTIMIZE (sec. 4.1.1.5)
- pragma PACK (sec. 4.4.6)
- pragma PAGE
- pragma PRIORITY
- pragma SHARED (sec. 4.1.1.6)
- pragma STORAGE_UNIT (sec. 4.1.1.4)
- pragma SUPPRESS
- pragma SYSTEM_NAME (sec. 4.1.1.4)

The following sections summarize the effects of and restrictions on certain predefined pragmas.

4.1.1.1. Pragma CONTROLLED

Access collections are not subject to automatic storage reclamation so pragma CONTROLLED has no effect. Space deallocated by means of Unchecked_Deallocation will be reused by the allocation of new objects.

4.1.1.2. Pragma INLINE

Pragma INLINE is supported as described in the LRM 6.3.2, with the following restrictions and clarifications:

- The body of the subprogram to be expanded inline must be compiled before the unit that calls the subprogram. If the call is compiled prior to the subprogram body, inline expansion of that call will not be performed. A warning is issued when a call is not inlined because the body has not been compiled.
- If a unit contains a call that results in inlined code, any subsequent recompilation of the body of the called subprogram will make the unit containing the inlined call obsolete.
- When inlining across libraries, the body of the subprogram to be inlined must be exported from a frozen root library (sec. 10.3.15 and 10.3.19).

APPENDIX F OF THE Ada STANDARD

- The optimization level, as set by the compiler command line option or an OPTIMIZE pragma, determines whether an attempt is made to obey a pragma INLINE (sec. 8.2). If the compilation containing a call to the subprogram named in an INLINE pragma is compiled at the minimum or low optimization level, (UNIX: -Op0 or -Op1; VMS: /optimize=minimum or /optimize=low) inlining will not be attempted for that call.
- Inlining may not be performed if the compiler determines that the subprogram to be inlined is too complex. Typical examples are subprograms that recursively call themselves, or whose objects are referenced by enclosing subprograms.

4.1.1.3. Pragma INTERFACE

Pragma INTERFACE is supported as described in LRM 13.9.

The syntax of the pragma is:

```
pragma INTERFACE(language_name, subprogram_name);
```

The pragma is placed just after the subprogram declaration and will be applied to all subprograms declared thus far with the name subprogram_name.

The pragma associates a particular calling sequence with a subprogram whose implementation is provided in the form of an object code module. The librarian interface subcommand<All hosts: interface> (sec. 10.3.22) must be used to identify the associated object code module.

The language_name may be Ada, Assembly, or C. Any other language_name will be accepted, but ignored, and the default language, Ada, will be used.

It is almost always necessary to use a pragma LINKAGE_NAME (sec. 4.1.2.1) for interfaced subprograms. Without the LINKAGE_NAME pragma, the user must determine the compressed name the compiler generates and use that name in the provided object module.

An interfaced subprogram cannot have a direct Ada implementation, i.e., a body is not allowed for such a subprogram. It is possible to compile an Ada subprogram with a different name and then use the librarian interface subcommand to reference that subprogram.

4.1.1.4. Pragas MEMORY_SIZE, STORAGE_UNIT, and SYSTEM_NAME

This section details the procedure for compiling one of these pragmas. The compilation unit containing the pragma must be compiled into a library that contains package System. For most users, the Tartan Ada Standard Packages Library will be the library that includes package System. In that case, the procedure is as follows: [This procedure will not cause any of the units in the Tartan Ada Standard Packages Library to become obsolete.]

1. Thaw the library tartan:standard_packages.
2. Compile the pragma into the library tartan:standard_packages. This step updates package System. Any unit that depends on System becomes obsolete and will require recompilation before it may be used in a program.

APPENDIX F OF THE Ada STANDARD

3. Freeze the library `tartan:standard_packages`.

For pragma `STORAGE_UNIT`, no value other than that already specified by `System.Storage_Unit` (sec. 4.3) is allowed. For pragma `SYSTEM_NAME`, no value other than that already specified by `System.System_Name` (sec. 4.3) is allowed.

4.1.1.5. Pragma `OPTIMIZE`

Pragma `OPTIMIZE` is supported as described in the LRM, Annex B with the following exceptions:

- pragma `OPTIMIZE` is not implemented for the declarative part of a block
- in the declarative part of bodies, the effect of pragma `OPTIMIZE` on nested subprograms is dependent on its position in the declarative part, i.e., the pragma applies to subprograms declared after it.

The argument applied to pragma `OPTIMIZE` (space or time) directly corresponds to the same argument supplied with the optimization option on the compiler command line. For example, specifying

```
pragma OPTIMIZE(TIME)
```

has the same effect as compiling the subprogram and specifying optimization level `time(UNIX: -Op3; VMS: /optimize=time)` on the command line.

When the code being compiled contains an `OPTIMIZE` pragma and the command line option to specify an optimization level is supplied, the pragma takes precedence over the command line option.

4.1.1.6. Pragma `SHARED`

Pragma `SHARED` is supported as described in the LRM, Annex B. Users should be aware that one consequence of applying pragma `SHARED` to a variable is to disable compiler optimizations that remove redundant reads and/or writes to that variable. Thus pragma `SHARED` allows the user to write loops that poll hardware devices until some change is seen.

```
with System;
package body Device_X is
  Control_Register_For_X : Integer;
  for Control_Register_For_X use at 16#123#;
  pragma SHARED(Control_Register_For_X);
  ...
  function Wait_Until_X_Signals return Integer is
    -- return the value of the control register as soon as it becomes
    -- non-zero
    T : Integer;
  begin
    loop
      T := Control_Register_For_X;
      exit when T /= 0;
    end loop;
    return T;
  end Wait_Until_X_Signals;
  ...
end Device_X;
```

Figure 4-1: Using pragma `SHARED` to Keep Redundant Reads

APPENDIX F OF THE Ada STANDARD

Note that pragma `SHARED` can be applied only to scalar variables. If the object is best described using a structure, one of the following alternatives must be used:

- Compile the code at low optimization levels, causing the compiler not to eliminate the redundant reads and writes. Since most applications cannot tolerate this kind of loss in performance across the board, it is suggested that the critical code be isolated into its own compilation unit.
- Declare the variable as a scalar (or adjacent scalars) and use unchecked conversion to copy to/from local variables of the proper type just after/before read/write operations.
- Use non-optimizable procedures that implement a read/write of the object. Package `Low_Level_IO` (LRM 14.6) provides one such set of routines. Other high-efficiency routines may be built using package `Machine_Code`.
- Another method is to create the obvious simple Ada routines, but isolate them in a separate package and do not use cross-compilation optimizations, such as pragma `INLINE`.

Figure 4-2 shows how package `Low_Level_IO` is used to force the compiler to keep redundant reads of a non-scalar object.

```
with Low_Level_IO;
with System;
package body Device_Y is
  ...
  type Y_Regs_Type is
    record
      Control : Integer;
      Input   : Integer;
      Output  : Integer;
    end record;
  Y : Y_Regs_Type;
  for Y use at 16#246#;
  ...
  function Wait_Until_Y_Signals return Integer is
    -- return the value of the control register as soon as it becomes
    -- non-zero
    T : Integer;
  begin
    loop
      Low_Level_IO.Receive_Control(Y.Control, T);
      exit when T /= 0;
    end loop;
    return T;
  end Wait_Until_Y_Signals;
  ...
end Device_Y;
```

Figure 4-2: Using `Low_Level_IO` to Force Compiler to Keep Redundant Reads

APPENDIX F OF THE Ada STANDARD

4.1.2. Implementation-Defined Pragmas

Implementation-defined pragmas provided by Tartan are described in the following sections.

4.1.2.1. Pragma LINKAGE_NAME

The pragma LINKAGE_NAME associates an Ada entity with a string that is meaningful externally; for example, to a linkage editor. It takes the form

```
pragma LINKAGE_NAME(name, string-constant)
```

The pragma is only allowed in a package specification or in a declarative part, or after a library subprogram in a compilation before any subsequent compilation unit.

If the pragma appears in a library package specification the name must denote an entity declared earlier in the same package. If the pragma appears in any other package specification or in a declarative part, the name must denote an entity declared earlier in the same package or declarative part, and must denote either a subprogram or an exception. If the pragma appears after a given library subprogram, the only name allowed is the name of this subprogram.

The name must be the simple name or operator symbol of an Ada entity. The name refers only to the most recently declared entity with the given name, not to all of the overloads of the name.

The name should denote an entity that has a runtime representation; for example, a subprogram, an exception, or an object. If the name denotes an entity that has no runtime representation the pragma has no effect; for example, named numbers, generic units, and most constants with values known at compile-time do not have runtime representations. The pragma also will have no effect if the name is one declared by a renaming declaration.

The effect of the pragma is to cause the string-constant to be used in the generated object code as an external name for the associated Ada entity. It is the responsibility of the user to guarantee that this string constant is meaningful to the linkage editor and that no illegal linkname clashes arise. Names given in the string-constant argument of a pragma LINKAGE_NAME are case sensitive. For example, `any_Old_LINKname` is not equivalent to `ANY_OLD_LINKNAME`. Therefore, a misspelled linkname will cause the link to fail.

When determining the maximum allowable length for the external linkage name, keep in mind that the compiler will generate names for elaboration flags for subprograms simply by appending a five-character suffix to the linkage name. Therefore, a linkage name for a subprogram may have five fewer characters than the lower limit of other tools that need to process the name (e.g., the Tartan Linker limits names to 40 characters; therefore, your external linkage name should not exceed 35 characters).

4.1.2.1.1. Calling Ada Subprograms from non-Ada Code

Pragma LINKAGE_NAME can be used to allow non-Ada code to call an Ada subprogram. Calling Ada from non-Ada code is highly dependent on the language the call is being made from as well as the compiler for that language.

APPENDIX F OF THE Ada STANDARD

First, the Ada subprogram must be given a linkage name so that the non-Ada code will be able to call the Ada subprogram. Pragma `LINKAGE_NAME` is used to perform this task. Next, an unimplemented subprogram must be defined in the non-Ada code. This subprogram must have the same linkage name as specified by the pragma `LINKAGE_NAME` on the Ada subprogram. Calls to this unimplemented subprogram will then go to the Ada subprogram if the object file containing the Ada subprogram was linked into the application. Note that a conversion between the object file (`file.tof`) for the Ada subprogram and the non-Ada object file will probably be needed. See the Object File Utilities Manual for more on object file conversion.

For the call to be made correctly, you must ensure that the calling convention and parameter passing mechanisms of the non-Ada code are compatible with the Tartan Ada Compiler (sec. 5.4 and 5.5).

Pragma `LINKAGE_NAME` can also be applied to an Ada object or exception. This allows non-Ada code to refer to the Ada entity. In doing so, you should be aware of the Ada rules governing implicit initialization of objects (Ada LRM 3.2.1 and 3.3) and the Tartan Ada Compiler's data representation conventions (sec. 5.1).

4.1.2.2. Pragma `FOREIGN_BODY`

In addition to pragma `INTERFACE`, Tartan Ada supplies pragma `FOREIGN_BODY` as a way to access entities defined in programs written in other languages. Use of the pragma `FOREIGN_BODY` dictates that all subprograms and objects in the package are provided by means of a foreign object module. Unlike pragma `INTERFACE`, pragma `FOREIGN_BODY` allows access to objects as well as subprograms.

The pragma is of the form:

```
pragma FOREIGN_BODY(language_name [, elaboration_routine_name])
```

A single such pragma may appear in any non-generic library package, and must appear in the visible part of the package before any declarations. The pragma is only permitted when the declarations in the visible and private parts of the package consist of subprogram declarations, number declarations, and object declarations with no explicit initialization and with a subtype given by a simple type mark. Use clauses and other pragmas may also appear in the package specification. If any of these restrictions are violated, the pragma is ignored and a warning is generated. Note in particular that types, exceptions, packages, and generic units may not be declared in the package.

The `language_name` argument is a string intended to identify the language processor used to create the foreign module. It is treated as a comment by the compiler.

The optional `elaboration_routine_name` argument is a string giving the linkage name of a routine to initialize the package. The routine specified will be called for the elaboration of this package body. It must be a global routine in the object module provided by the user.

The programmer must ensure that the calling convention and data representation of the foreign body subprograms and elaboration routine are compatible with those used by the Tartan Ada Compiler (sec. 5.4).

APPENDIX F OF THE Ada STANDARD

To successfully link a program including a foreign body, the object module for that body must be provided to the library using the librarian foreign subcommand<UNIX: foreign; VMS: foreign> (sec. 2.3.3 and 10.3.18).

All entities declared by the package must be supplied by the foreign object module. Pragma LINKAGE_NAME will usually have to be used to ensure agreement between the linkage names used by the Tartan Ada Compiler and the foreign language processor.

The foreign body is entirely responsible for initializing objects declared in a package utilizing pragma FOREIGN_BODY. In particular, the user should be aware that the implicit initializations described in LRM 3.2.1 are not done by the compiler. (These implicit initializations are associated with objects of access types, certain record types and composite types containing components of the preceding kinds of types.)

The user may choose to override the pragma FOREIGN_BODY and compile a corresponding package body written in Ada. In this case the pragma is ignored (in particular the specified elaboration routine is not called), and no librarian foreign subcommand is required or allowed. This capability is useful for rapid prototyping, where an Ada package may serve to provide a simulated response for the functionality that a foreign body may eventually produce. It also allows the user to replace a foreign body with an Ada body without recompiling the specification.

If only subprograms are declared in the package specification it is more portable to use pragma INTERFACE on each of the subprograms instead of pragma FOREIGN_BODY on the package.

In the following example, we want to call a function plmn which computes polynomials and is written in assembler.

```
package Math_Functions is

  pragma FOREIGN_BODY("assembler");
  function Polynomial(X:Integer) return Integer;
  -- Ada spec matching the assembler routine
  pragma LINKAGE_NAME(Polynomial, "plmn");
  -- force compiler to use name plmn when referring to this function

end Math_Functions;

with Math_Functions; use Math_Functions;
procedure Main is
  X:Integer := Polynomial(10);
  -- will generate a call to plmn
  begin ...
end Main;
```

To compile, link and run the above program, you must:

1. Compile Math_Functions.
2. Compile Main.
3. Provide the object module (for example, math.tof) containing the assembled code for plmn, converted to Tartan Object File Format (TOFF)

APPENDIX F OF THE Ada STANDARD

using the ITS-to-TOFF utility (Object File Utilities Manual, ch. 4).

4. Issue the command:

UNIX:

```
adalib1750a foreign math_functions math.tof
```

VMS:

```
all7 foreign math_functions math.tof
```

5. Issue the command:

UNIX:

```
adalib1750a link main
```

VMS:

```
all7 link main
```

Without step 4, an attempt to link will produce an error message informing you of a missing package body for Math_Functions.

4.1.2.3. Pragma UNCHECKED_NO_STATE_WRITTEN and Pragma UNCHECKED_NO_STATE_WRITTEN_OR_READ

The pragmas UNCHECKED_NO_STATE_WRITTEN and UNCHECKED_NO_STATE_WRITTEN_OR_READ take the form:

```
pragma UNCHECKED_NO_STATE_WRITTEN(name [, name...])  
pragma UNCHECKED_NO_STATE_WRITTEN_OR_READ(name [, name...])
```

Each name must be the simple name of an Ada subprogram declared in the declarative part or package specification where the pragma appears. The name refers only to the most recently declared subprogram with the given name, not to all of the overloads of the name.

The pragma UNCHECKED_NO_STATE_WRITTEN notifies the compiler that the named subprogram has no side effects on any objects outside the subprogram. Assignment to in out or out parameters is not considered a side effect. Function results are also not considered to be side effects. Calling another subprogram is considered to be a side effect, unless the called subprogram is also named in either a pragma UNCHECKED_NO_STATE_WRITTEN or pragma UNCHECKED_NO_STATE_WRITTEN_OR_READ.

This pragma permits the compiler to improve the optimization performed near calls to the named subprogram without introducing a dependency on the body of the subprogram. In effect, global side effect analysis is achieved without creating additional dependencies which may require recompilation.

Any function which writes only to its result, or any subprogram which writes only to its in out or out parameters is an excellent candidate for this pragma.

APPENDIX F OF THE Ada STANDARD

The pragma `UNCHECKED_NO_STATE_WRITTEN_OR_READ` indicates that the named subprogram behaves strictly as a mathematically pure function. In essence, this statement means that the subprogram will always return the same result when called with identical parameters. The named subprogram must follow all of the rules for an `UNCHECKED_NO_STATE_WRITTEN` subprogram. In addition, the named subprogram may not read the value of any variable not contained within its own scope, in and in out parameters, and objects declared as constants may be read freely. Called subprograms must themselves be `NO_STATE_WRITTEN_OR_READ`.

The compiler may choose to make common subexpressions of the results of calls to the named subprogram. It may also remove such calls entirely when the result of the subprogram is not used; calls may also be loop-invariant hoisted.

Subprograms which are likely candidates for this pragma include math package subprograms, matrix math subprograms, trigonometric functions, etc.
Caution:

Pragmas `UNCHECKED_NO_STATE_WRITTEN` and `UNCHECKED_NO_STATE_WRITTEN_OR_READ` are strictly assertive in nature and are entirely unchecked. The compiler will not notify you if the body of the subprogram does not meet the requirements of the pragma. When one of these pragmas is incorrectly applied to a subprogram which does not meet its requirements, the behavior of your program is undefined and may be unpredictable.

4.2. IMPLEMENTATION-DEPENDENT ATTRIBUTES

4.2.1. 'Exception_Address

The attribute `'Exception_Address` used with a prefix that denotes an exception yields the storage address associated with the exception. The value of this attribute is of the type `Address` defined in the package `System`.

4.2.2. 'Physical_Address

The attribute `'Physical_Address` used with a prefix that denotes an object, a program unit, or a label yields the physical address of the object or of the code associated with the program unit or label. The value of the attribute is of the type `Physical_Address` defined in the package `System`.

The `Physical_Address` attribute is meaningful only for a prefix which denotes a statically allocated entity, i.e., the address is known at link-time.

When no meaningful result can be returned by the attribute, the value of the attribute will be zero and a compiler warning message will be issued.

4.3. SPECIFICATION OF THE PACKAGE `System`

The parameter values specified for MIL-STD-1750A in package `System` (LRM 13.7.1 and Annex C) are:

APPENDIX F OF THE Ada STANDARD

```
package System is
  type Address is new Integer;
  type Physical_Address is new Long_Integer;
  type Name is (MIL_STD_1750A);
  System_Name : constant Name := MIL_STD_1750A;
  Storage_Unit : constant := 16;
  Memory_Size : constant := 65536;
  Max_Int : constant := 2_147_483_647;
  Min_Int : constant := -Max_Int - 1;
  Max_Digits : constant := 9;
  Max_Mantissa : constant := 31;
  Fine_Delta : constant := 2#1.0#e-31;
  Tick : constant := 0.0001;
  subtype Priority is Integer range 10 .. 200;
  Default_Priority : constant Priority := Priority'First;
  Runtime_Error : exception;
end System;
```

4.4. RESTRICTIONS ON REPRESENTATION CLAUSES

The following sections explain the basic restrictions for representation specifications followed by additional restrictions applying to specific kinds of clauses.

4.4.1. Basic Restriction

The basic restriction on representation specifications (LRM 13.1) is that they may be given only for types declared in terms of a type definition, excluding a *Generic_Type_Definition* (LRM 12.1) and a *Private_Type_Definition* (LRM 7.4). Any representation clause in violation of these rules is not obeyed by the compiler; an error message is issued.

Further restrictions are explained in the following sections. Any representation clauses violating those restrictions cause compilation to stop and a diagnostic message to be issued.

4.4.2. Length Clauses

Length clauses (LRM 13.2) are, in general, supported. The following sections detail use and restrictions.

4.4.2.1. Size Specifications for Types

The rules and restrictions for size specifications applied to types of various classes are described below.

The following principle rules apply:

1. The size is specified in bits and must be given by a static expression.
2. The specified size is taken as a mandate to store objects of the type in the given size wherever feasible. No attempt is made to store values of the type in a smaller size, even if possible. The following rules apply with regard to feasibility:

APPENDIX F OF THE Ada STANDARD

- An object that is not a component of a composite object is allocated with a size and alignment that is referable on the target machine (i.e., no attempt is made to create objects of non-referable size on the stack). If such stack compression is desired, it can be achieved by the user by combining multiple stack variables in a composite object; for example:

```
type My_Enum is (A, B);
for My_Enum'Size use 1;
V, W : My_Enum; -- will occupy two storage
                -- units on the stack
                -- (if allocated at all)
type Rec is record
  V, W : My_Enum;
end record;
pragma PACK(Rec);
O : Rec;        -- will occupy one storage unit
```

- A formal parameter of the type is sized according to calling conventions rather than size specifications of the type. Appropriate size conversions upon parameter passing take place automatically and are transparent to the user.
- Adjacent bits to an object that is a component of a composite object, but whose size is non-referable, may be affected by assignments to the object, unless these bits are occupied by other components of the composite object (i.e., whenever possible, a component of non-referable size is made referable).

In all cases, the compiler generates correct code for all operations on objects of the type, even if they are stored with differing representational sizes in different contexts.

Note: A size specification cannot be used to force a certain size in value operations of the type; for example:

```
type My_Int is range 0..65535;
for My_Int'Size use 16; -- o.k.
A, B : My_Int;
...A + B... -- this operation will generally be
            -- executed on 32-bit values
```

3. A size specification for a type specifies the size for objects of this type and of all its subtypes. For components of composite types, whose subtype would allow a shorter representation of the component, no attempt is made to take advantage of such shorter representations.

For example, consider the following:

```
type My_Int is range 0..2**17-1;
for My_Int'Size use 17; -- (1)
subtype Small_My_Int is My_Int range 0..255;
type R is record
  ...
  X : Small_My_Int;
  ...
end record;
```

APPENDIX F OF THE Ada STANDARD

The component R.X will occupy 17 bits even though it can be represented in 8 bits. If a pragma PACK(R) is added, R.X will still be allocated in 17 bits.

In contrast, for types without a size specification, such components may be represented in a lesser number of bits than the number of bits required to represent all values of the type. In the example above, if the size specification at (1) is removed, R.X will be represented in 32 bits (the size of My_Int). However, a pragma PACK(R) will now cause R.X to be allocated in 8 bits.

Size specifications for access types must coincide with the default size chosen by the compiler for the type.

Size specifications are not supported for floating-point types or task types.

No useful effect can be achieved by using size specifications for access, floating-point, or task types.

4.4.2.2. Size Specification for Scalar Types

The specified size must accommodate all possible values of the type including the value 0 (zero), even if 0 is not in the range of the values of the type. For numeric types with negative values, the number of bits must account for the sign bit. Biased representation is not attempted. Thus,

```
type My_Int is range 100..101;
```

requires at least 7 bits, although it has only two values, while

```
type My_Int is range -101..-100;
```

requires 8 bits to account for the sign bit.

A size specification for a fixed-point type does not affect the accuracy of operations on the type. Such influence should be exerted via the Accuracy_Definition of the type (LRM 3.5.9).

A size specification for a scalar type may not specify a size larger than the largest operation size supported by the target architecture for the respective class of values of the type.

4.4.2.3. Size Specification for Array Types

A size specification for an array type must be large enough to accommodate all components of the array under the densest packing strategy. Any alignment constraints on the component type (sec. 4.4.7) must be met.

Arrays with component size less than or equal to 16 bits are densely packed. No pad or unused bits exist between components within a word. Arrays with component size greater than 16 bits are padded up to the next 16-bit boundary. The size of the component type cannot be influenced by a length clause for an array. Within the limits of representing all possible values of the component subtype (but not necessarily of its type), the representation of components may, however, be reduced to the minimum number of bits, unless the component type carries a size specification.

APPENDIX F OF THE Ada STANDARD

If there is a size specification for the component type, but not for the array type, the component size is rounded up to a referable size, unless pragma PACK is given. This rule applies even to boolean types or other types that require only a single bit for the representation of all values.

4.4.2.4. Size Specification for Record Types

A size specification for a record type does not influence the default type mapping of a record type. The size must be at least as large as the number of bits determined by type mapping. Influence over packing of components can be exerted by means of (partial) record representation clauses or by pragma PACK.

Neither the size of component types, nor the representation of component subtypes can be influenced by a length clause for a record.

The only implementation-dependent components allocated by Tartan Ada in records contain either dope information for arrays whose bounds depend on discriminants of the record or relative offsets of components within a record layout for record components of dynamic size. These implementation-dependent components cannot be named or sized by the user.

A size specification cannot be applied to a record type with components of dynamically determined size.

Note: Size specifications for records can be used only to widen the representation accomplished by padding at the beginning or end of the record. Any narrowing of the representation over default type mapping must be accomplished by representation clauses or pragma PACK.

4.4.2.5. Specification of Collection Sizes

The specification of a collection size causes the collection to be allocated with the specified size. It is expressed in storage units and need not be static; refer to package System for the meaning of storage units.

Any attempt to allocate more objects than the collection can hold causes a Storage_Error exception to be raised. Dynamically sized records or arrays may carry hidden administrative storage requirements that must be accounted for as part of the collection size. Moreover, alignment constraints on the type of the allocated objects may make it impossible to use all memory locations of the allocated collection. No matter what the requested object size, the allocator must allocate a minimum of 2 words per object. This lower limit is necessary for administrative overhead in the allocator. For example, a request of 5 words results in an allocation of 5 words; a request of 1 (one) word results in an allocation of 2 words.

In the absence of a specification of a collection size, the collection is extended automatically if more objects are allocated than possible in the collection originally allocated with the compiler-established default size. In this case, Storage_Error is raised only when the available target memory is exhausted. If a collection size of zero is specified, no access collection is allocated.

Collection sizes may not be specified for an access type whose designated type is a task type.

APPENDIX F OF THE Ada STANDARD

4.4.2.6. Specification of Task Activation Size

The specification of a task activation size causes the task activation to be allocated with the specified size. It is expressed in storage units; refer to package System for the meaning of storage units.

Any attempt to exceed the activation size during execution causes a Storage_Error exception to be raised. Unlike collections, there is no extension of task activations.

4.4.2.7. Specification of 'Small

Only powers of 2 are allowed for 'Small.

The length of the representation may be affected by this specification. If a size specification is also given for the type, the size specification takes precedence; it must then be possible to accommodate the specification of 'Small within the specified size.

4.4.3. Enumeration Representation Clauses

For enumeration representation clauses (LRM 13.3), the following restrictions apply:

- The internal codes specified for the literals of the enumeration type may be any integer value between Long_Integer'First and Long_Integer'Last. It is strongly advised that you do not provide a representation clause that merely duplicates the default mapping of enumeration types which assigns consecutive numbers in ascending order starting with 0 (zero). Unnecessary runtime cost is incurred by such duplication. It should be noted that the use of attributes on enumeration types with user-specified encodings is costly at runtime.
- Array types, whose index type is an enumeration type with non-contiguous value encodings, consist of a contiguous sequence of components. Indexing into the array involves a runtime translation of the index value into the corresponding position value of the enumeration type.

4.4.4. Record Representation Clauses

The alignment clause of record representation clauses (LRM 13.4) is observed for library-level record objects.

The alignment clause has no effect on objects not defined on a library level (e.g., subprogram locals). The value given in the alignment clause is in addressable units and is restricted to the powers of two in the range of

0 15
2 ..2 . The specified alignment becomes the minimum alignment of the record type, unless the minimum alignment of the record forced by the component allocation is already more stringent than the value specified by the alignment clause.

The component clauses of record representation clauses are allowed only for components and discriminants of statically determinable size. Not all components need to be present. Component clauses for components of variant parts are allowed only if the size of the record type is statically determinable for every variant.

APPENDIX F OF THE Ada STANDARD

The size specified for each component must be sufficient to allocate all possible values of the component subtype, but not necessarily the component type. The location specified must be compatible with any alignment constraints of the component type; an alignment constraint on a component type may cause an implicit alignment constraint on the record type itself.

If some, but not all, discriminants and components of a record type are described by a component clause, the discriminants and components without component clauses are allocated after those with component clauses; no attempt is made to utilize gaps left by the user-provided allocation.

4.4.5. Address clauses

Address clauses (LRM 13.5) are supported with the following restrictions:

- When applied to an object, an address clause becomes a linker directive to allocate the object at the given address. For any object not declared immediately within a top-level library package, the address clause is meaningless.
- Address clauses applied to local packages are not supported by Tartan Ada. Address clauses applied to library packages are prohibited by the syntax; therefore, an address clause can be applied to a package only if it is a body stub.
- Address clauses applied to subprograms and tasks are implemented according to the LRM rules. When applied to an entry, the specified value identifies an interrupt in a manner customary for the target. Immediately after a task is created, a runtime call is made for each of its entries having an address clause, establishing the proper binding between the entry and the interrupt.
- A specified address must be an Ada static expression, as defined in LRM 4.9.
- Address clauses which are applied to objects, subprograms, packages, or task units specify virtual addresses.
- The range of System.Address is -16#8000#..16#7fff#. To represent a machine virtual address in the range 16#0000#..16#7fff#, use the corresponding System.Address. To represent a machine virtual address greater than System.Address 16#7fff#, use the negated radix-complement of the desired machine virtual address. For example, to express machine virtual address 16#C000#, use 16#C000#-2**16.

Note: Creating an overlay of two objects by means of address clauses is possible with Tartan Ada. However, such overlays (which are considered erroneous by the Ada LRM 13.5(8)) will not be recognized by the compiler as an aliasing that prevents certain optimizations. Therefore, problems may arise if reading and writing of the two overlaid objects are intermingled. For example, if variables A and B are overlaid by means of address clauses, the Ada code sequence:

```
A := 5;
B := 7;
if A = 5 then raise Surprise; end if;
```

APPENDIX F OF THE Ada STANDARD

may well raise the exception Surprise, since the compiler believes the value of A to be 5 even after the assignment to B.

4.4.6. Pragma PACK

Pragma PACK (LRM 13.1) is supported. For details, refer to the following sections.

4.4.6.1. Pragma PACK for Arrays

If pragma PACK is applied to an array, the densest possible representation is chosen. For details of packing, refer to the explanation of size specifications for arrays (sec. 4.4.2.3).

If, in addition, a length clause is applied to the array type, the pragma has no effect, since such a length clause already uniquely determines the array packing method.

If a length clause is applied to the component type, the array is packed densely, observing the component's length clause. Note that the component length clause may have the effect of preventing the compiler from packing as densely as would be the default if pragma PACK were applied with no length clause given for the component type.

4.4.6.2. The Predefined Type String

Package Standard applies pragma PACK to the type String. However, when applied to character arrays, this pragma cannot be used to achieve denser packing than is the default for the target: 1 character per 16-bit word.

4.4.6.3. Pragma PACK for Records

If pragma PACK is applied to a record, the densest possible representation is chosen that is compatible with the sizes and alignment constraints of the individual component types. Pragma PACK has an effect only if the sizes of some component types are specified explicitly by size specifications and are non-referable. In the absence of pragma PACK, such components generally consume a referable amount of space.

It should be noted that the default type mapping for records maps components of boolean or other types that require only a single bit to a single bit in the record layout, if there are multiple such components in a record. Otherwise, it allocates a referable amount of storage to the component.

If pragma PACK is applied to a record for which a record representation clause has been given detailing the allocation of some but not all components, the pragma PACK affects only the components whose allocation has not been detailed. Moreover, the strategy of not utilizing gaps between explicitly allocated components still applies.

4.4.7. Minimal Alignment for Types

Certain alignment properties of values of certain types are enforced by the type mapping rules. Any representation specification that cannot be satisfied within these constraints is not obeyed by the compiler and is appropriately diagnosed.

APPENDIX F OF THE Ada STANDARD

Alignment constraints are caused by properties of the target architecture, most notably by the capability to extract non-aligned component values from composite values in a reasonably efficient manner. Typically, restrictions exist that make extraction of values that cross certain address boundaries very expensive, especially in contexts involving array indexing. Permitting data layout that require such complicated extractions may impact code quality on a broader scale than merely in the local context of such extractions.

Instead of describing the precise algorithm of establishing the minimal alignment of types, we provide the general rule that is being enforced by the alignment rules:

- No object of scalar type (including components or subcomponents of a composite type) may span a target-dependent address boundary that would mandate an extraction of the object's value to be performed by two or more extractions.

4.5. IMPLEMENTATION-GENERATED COMPONENTS IN RECORDS

The only implementation-dependent components allocated by Tartan Ada in records are fields containing either dope information for arrays whose bounds depend on discriminants of the record or relative offsets of components within a record layout for record components of dynamic size. These components cannot be named by the user.

4.6. INTERPRETATION OF EXPRESSIONS APPEARING IN ADDRESS CLAUSES

Section 13.5.1 of the LRM describes a syntax for associating interrupts with task entries. Tartan Ada implements the address clause

for ToEntry use at intID;

by associating the interrupt specified by intID with the ToEntry entry of the task containing this address clause. The interpretation of intID is implementation dependent.

The 1750A Ada runtimes provide 16 interrupts that may be associated with task entries. These interrupts are identified by an integer in the range 0..15. The intID argument of an address clause is interpreted as follows:

- If the argument is in the range 0..15, a full support interrupt association is made between the interrupt specified by the argument and the task entry.
- If the argument is in the range 16..31, a fast interrupt association is made between the interrupt number (argument-16) and the task entry.
- If the argument is outside the range 0..31, the program is erroneous.

For the difference between full support and fast interrupt handling, refer to section 9.5.6.

4.7. RESTRICTIONS ON UNCHECKED CONVERSIONS

Tartan supports Unchecked_Conversion as documented in Section 13.10 of the LRM. The sizes need not be the same, nor need they be known at compile time. The only exception is unconstrained array types or access to unconstrained array types which may not be used as the target of an Unchecked_Conversion.

APPENDIX F OF THE Ada STANDARD

If the value in the source is wider than that in the target, the source value will be truncated. If narrower, it will be zero-extended. Calls on instantiations of `Unchecked_Conversion` are made inline automatically.

4.8. IMPLEMENTATION-DEPENDENT ASPECTS OF INPUT-OUTPUT PACKAGES

Tartan Ada supplies the predefined input/output packages `Direct_IO`, `Sequential_IO`, `Text_IO`, and `Low_Level_IO` as required by LRM Chapter 14. However, since MIL-STD-1750A is used in embedded applications lacking both standard I/O devices and file systems, the functionality of `Direct_IO`, `Sequential_IO`, and `Text_IO` is limited.

`Direct_IO` and `Sequential_IO` raise `Use_Error` if a file open or file access is attempted. `Text_IO` is supported to `Current_Output` and from `Current_Input`. A routine that takes explicit file names raises `Use_Error`. `Low_Level_IO` for MIL-STD-1750A provides an interface by which the user may execute XIO operations. The Ada specifications for the Tartan Ada standard packages are located online in the directory:

UNIX: `/installation_directory/1750a/version/std_packages/src`
VMS: `[installation_directory.1750a.version.std_packages.src]` In both the `Send_Control` and `Receive_Control` procedures, the device parameter specifies an XIO address while the data parameter is the single word of data transferred.

4.9. OTHER IMPLEMENTATION CHARACTERISTICS

The following information is supplied in addition to that required by Appendix F to MIL-STD-1815A.

4.9.1. Definition of a Main Program

Any Ada library subprogram unit may be designated the main program for purposes of linking (using the Ada librarian's link subcommand) provided that the subprogram has no parameters.

Tasks initiated in imported library units follow the same rules for termination as other tasks (described in LRM 9.4 (6-10)). Specifically, these tasks are not terminated simply because the main program has terminated. Terminate alternatives in selective wait statements in library tasks are therefore strongly recommended.

4.9.2. No Use of `Numeric_Error`

No predefined operations will raise the exception `Numeric_Error`. The Tartan Ada Compiler raises the predefined exception `Constraint_Error` in situations where, according to the Ada LRM, the predefined exception `Numeric_Error` should be raised.

This change in the compiler has been made in accordance with the approved Ada Interpretation AI-00387. ``An Ada program is portable if any handler intended to process `Numeric_Error` provides a choice for `Constraint_Error` and processes it in a similar manner.''

If this procedure has been followed, you will not experience any change in program behavior because of this change in the compilation system.

4.9.3. Implementation of Generic Units

APPENDIX F OF THE Ada STANDARD

All instantiations of generic units, except the predefined generic `Unchecked_Conversion` and `Unchecked_Deallocation` subprograms, are implemented by code duplications. No attempt at sharing code by multiple instantiations is made in this release of Tartan Ada.

Tartan Ada enforces the restriction that the body of a generic unit must be compiled before the unit can be instantiated. It does not impose the restriction that the specification and body of a generic unit must be provided as part of the same compilation. A recompilation of the body of a generic unit will cause any units that instantiated this generic unit to become obsolete.

4.9.4. Implementation-Defined Characteristics in Package Standard

The implementation-dependent characteristics for MIL-STD-1750A in package `Standard` [Annex C] are:

```
package Standard is
...
type Integer is range -32768 .. 32767;
type Float is digits 6 range -16#0.8000_00#E+32 .. 16#0.7FFF_FF#E+32;
type Long_Integer is range -2147483648 .. 2147483647;
type Long_Float is digits 9 range -16#0.8000_0000_00#E+32 ..
    16#0.7FFF_FFFF_FF#E+32 ;
type Duration is delta 0.0001 range -86400.0 .. 86400.0;
    -- Duration'Small = 2#1.0#E-14
...
end Standard;
```

4.9.5. Attributes of Type Duration

The type `Duration` is defined with the following characteristics:

Attribute	Value
<code>Duration'Delta</code>	0.0001 sec
<code>Duration'Small</code>	6.103516E ⁻⁵ sec
<code>Duration'First</code>	-86400.0 sec
<code>Duration'Last</code>	86400.0 sec

APPENDIX F OF THE Ada STANDARD

4.9.6. Values of Integer Attributes

Tartan Ada supports the predefined integer types `Integer` and `Long_Integer`.

The range bounds of the predefined type `Integer` are:

Attribute	Value
<code>Integer'First</code>	-2^{**15}
<code>Integer'Last</code>	$2^{**15}-1$

The range bounds of the predefined type `Long_Integer` are:

Attribute	Value
<code>Long_Integer'First</code>	-2^{**31}
<code>Long_Integer'Last</code>	$2^{**31}-1$

The range bounds for subtypes declared in package `Text_IO` are:

Attribute	Value
<code>Count'First</code>	0
<code>Count'Last</code>	<code>Integer'Last - 1</code>

APPENDIX F OF THE Ada STANDARD

Positive_Count'First	1
Positive_Count'Last	Integer'Last - 1
Field'First	0
Field'Last	240

The range bounds for subtypes declared in packages Direct_IO are:

Attribute	Value
Count'First	0
Count'Last	Integer'Last/Element_Type'Size
Positive_Count'First	1
Positive_Count'Last	Count'Last

4.9.7. Values of Floating-Point Attributes

Attribute	Value for Float
Digits	6
Mantissa	21

APPENDIX F OF THE Ada STANDARD

Emax	84
Epsilon	16#0.1000_000#E-4 (approximately 9.53674E-07)
Small	16#0.8000_000#E-21 (approximately 2.58494E-26)
Large	16#0.FFFF_F80#E+21 (approximately 1.93428E+25)
Safe_Emax	127
Safe_Small	16#0.1000_000#E-31 (approximately 2.93874E-39)
Safe_Large	16#0.7FFF_FC0#E+32 (approximately 1.70141E+38)
First	-16#0.8000_000#E+32 (approximately -1.70141E+38)
Last	16#0.7FFF_FF0#E+32 (approximately 1.70141E+38)
Machine_Radix	2
Machine_Mantissa	23

APPENDIX F OF THE Ada STANDARD

Machine_Emax	127
Machine_Emin	-128
Machine_Rounds	true
Machine_Overflows	true

Attribute	Value for Long_Float
Digits	9
Mantissa	31
Emax	124
Epsilon	16#0.4000_0000_00#E-7 (approximately 9.3132257461548E-10)
Small	16#0.8000_0000_00#E-31 (approximately 2.3509887016445E-38)
Large	16#0.FFFF_FFFE_00#E+31 (approximately 2.1267647922655E+37)
Safe_Emax	127

APPENDIX F OF THE Ada STANDARD

Safe_Small	16#0.1000_0000_00#E-31 (approximately 2.9387358770557E-39)
Safe_Large	16#0.7FFF_FFFF_00#E+32 (approximately 1.7014118338124E+38)
First	-16#0.8000_0000_00#E+32 (approximately -1.7014118346016E+38)
Last	16#0.7FFF_FFFF_FF#E+32 (approximately 1.7014118346047E+38)
Machine_Radix	2
Machine_Mantissa	39
Machine_Emax	127
Machine_Emin	-128
Machine_Rounds	true
Machine_Overflows	true

4.10. SUPPORT FOR PACKAGE Machine_Code

Package Machine_Code provides an interface through which a user may request the generation of 1750A assembly instructions. The Tartan implementation of package Machine_Code is similar to that described in section 13.8 of the Ada LRM, with several added features. The Ada specifications for the Tartan Ada standard packages are located online in the directory:

UNIX: /installation_directory/1750a/version/std_packages/src
VMS: [installation_directory.1750a.version.std_packages.src]

APPENDIX F OF THE Ada STANDARD

4.10.1. Basic Information

As required by LRM, section 13.8, a routine which contains machine code inserts may not have any other kind of statement, and may not contain an exception handler. The only allowed declarative item is a use clause. Comments and pragmas are allowed as usual.

4.10.2. Instructions

A machine code insert has the form `Type_Mark'Record_Aggregate`, where the type must be one of the records defined in package `Machine_Code`. Package `Machine_Code` defines three types of records. Each has an opcode and zero, one or two operands. These records allow for the expression of the entire 1750A Assembly language.

4.10.3. Operands and Address Modes

An operand consists of a record aggregate which contains the information needed to specify the corresponding Assembly instruction for generation by the compiler.

Each operand in a machine code insert must have an `Address_Mode_Name`. The address modes specified in package `Machine_Code` are sufficient to provide all address modes supported by the 1750A Assembly language.

In addition, package `Machine_Code` supplies the address modes `Symbolic_Address` and `Symbolic_Value` which allow the user to refer to Ada objects by specifying `Object'Address` as the value for the operand. Any Ada object which has the `'Address` attribute may be used in a symbolic operand. `Symbolic_Address` should be used when the operand is a true address (i.e., a branch target for example). `Symbolic_Value` should be used when the operand is actually a value (i.e., one of the operands of an AR instruction).

When an Ada object is used as a source operand in an instruction (that is, one from which a value is read), the compiler will generate code which fetches the value of the Ada object. When an Ada object is used as the destination operand of an instruction, the compiler will generate code which uses the address of the Ada object as the destination of the instruction.

For source operands:

- `Symbolic_Address` means that the address specified by the `'Address` expression is used as the source bits. When the Ada object specified by the `'Address` instruction is bound to a register, this will cause a compile-time error message because it is not possible to "take the address" of a register.
- `Symbolic_Value` means that the value found at the address specified by the `'Address` expression will be used as the source bits. An Ada object which is bound to a register is correct here, because the contents of a register can be expressed on the 1750A.
- `PcRel` indicates that the address of the label will be used as the source bits.
- Any other non-register means that the value found at the address specified by the operand will be used as the source bits.

APPENDIX F OF THE Ada STANDARD

For destination operands:

- Symbolic_Address means that the desired destination for the operation is the address specified by the 'Address expression. An Ada object which is bound to a register is correct here; a register is a legal destination on the 1750A.
- Symbolic_Value means that the desired destination for the operations is found by fetching 16 bits from the address specified by the 'Address expression, and storing the result to the address represented by the fetched bits. This is equivalent to applying one extra indirection to the address used in the Symbolic_Address case.
- All other operands are interpreted as directly specifying the destination for the operation.

When an Ada object is used as a source operand in an instruction, the compiler generates code which fetches the value of the Ada object. When an Ada object is used as the destination operand of an instruction, the compiler generates code which uses the address of the Ada object as the destination for the instruction.

4.10.4. Examples

The Tartan implementation of package Machine_Code makes it possible to specify both simple machine code inserts such as

```
TWO_OPNDS'(LR, (R_AM,R0), (R_AM,R1))
```

or more complex inserts such as

```
TWO_OPNDS'(AIM, (R_AM,R0), (Symbolic_Address, ARRAY_VAR(X,Y,27)'ADDRESS));
```

In the first example, the compiler emits the instruction LR R0,R1. In the second example, the compiler first emits whatever instructions are needed to form the address of ARRAY_VAR(X,Y,27) and then emits the AIM instruction. The various error checks specified in the LRM will be performed on all compiler-generated code unless they are suppressed by the programmer (either through pragma SUPPRESS, or through command line options).

4.10.5. Incorrect Operands

Under some circumstances, the compiler attempts to correct incorrect operands. Three modes of operation are supplied for package Machine_Code to determine whether corrections are attempted and how much information about the necessary corrections is provided to the user.

The compiler command line options for the three modes of operation are:

APPENDIX F OF THE Ada STANDARD

UNIX	VMS
Me	fixup=none
Mw	fixup=warn
no option [default]	fixup=quiet [default]

In the Me or fixup=none mode, the compiler does not attempt to fix any mistakes that the programmer may have made in writing machine code inserts. The compiler simply issues error messages to flag the illegal code and creates a corresponding listing file. The most common errors are illegal register and addressing mode uses in the operand field of the array aggregate.

In the default or fixup=quiet mode, if incorrect addressing modes are chosen by the user in the array aggregates operand field, in most cases the compiler will be able to emit code that corrects the mistake by adding extra instructions or modifying the given instructions to create correct assembly code.

For example, although it is illegal to use a memory address as the source operand for an AR instruction, in the default or fixup=quiet mode, the compiler generates semantically correct code which tries to create the desired effect by adding new instructions in which user code

```
TWO_OPNDS' (AR, (R_AM,R3), (D_AM,OBJECT'ADDRESS));
```

produced output

```
L      R0,OBJECTADDRESS
AR     R3,R0
```

No warnings are issued to inform the user that substitutions have been made or that R0 has been used.

Another example is the use of address mode ``Direct'' which requires a type System.Address. When the user specifies a register, the following user code

```
TWO_OPNDS' (A, (R_AM,R1), (R_AM,R2));
```

produces output

```
STB    R14,0
A      R1,0,R14
```

Here the value of register R2 has been stored onto the program stack and then referenced as a memory location, which satisfies the requirement that an

APPENDIX F OF THE Ada STANDARD

address be used as the source operand for the ADD instruction.

In the Mw or fixup=warn mode, the compiler does its best to fix any incorrect operands for an instruction but also issues a warning message stating that the machine code insert required additional machine instructions to satisfy the 1750A Assembly language requirements.

4.10.6. Register Usage

Since the compiler may need to allocate registers as temporary storage in machine code routines, there are some restrictions placed on register usage; the compiler automatically frees all of the registers which are volatile across calls for your use (i.e., R0, R1, R2, and R3). In most instances if you reference any other registers, the compiler reserves them for your use until the end of the machine code routine. However, bear in mind that the compiler does not save the registers automatically if the routine is expanded inline. This means that the first reference to a register which is not volatile across calls should be an instruction which saves the register's contents to insure that the value is not overwritten and can later be restored at the end of the routine (by the user). This rule will help ensure correct operation of your machine code inserts even when inline expansion is performed by another routine (and possibly another user).

As a result of freeing all volatile registers for the user, any parameters which were passed in registers are moved to either non-volatile registers or to memory. References to parameter Address in machine code inserts will then produce code that accesses these register or memory locations. This means that there is a possibility of invalidating the value of some Address expressions if the non-volatile register to which the value is bound, is referenced as a destination in some later machine code insert. In this case, any subsequent references to the Address expression will cause the compiler to issue a warning message to this effect. When compiling in the UNIX default or Mw modes, or in the VMS fixup=quiet or fixup=warn modes, the compiler does its best to fix any incorrect operands for an instruction but also issues a warning message stating that the machine code insert required additional machine instructions to satisfy the 1750A Assembly language requirements. The compiler uses registers to effect the corrections. If you use all fifteen registers, corrections will not be possible. In general, when more registers are available to the compiler it is better able to generate efficient code.

4.10.7. Data Directives

Three special instructions are included in package Machine_Code to allow the user to place a bit pattern into the code stream. These instructions are DATA16, DATA32, and DATA48. Each of these instructions can have from 1 to 6 operands.

- DATA16 places 16-bit values into the code stream. Operands may include integers and addresses.
- DATA32 places 32-bit values into the code stream. Operands may include integers or floats.
- DATA48 places 48-bit values into the code stream. The only legal operands are floats.

APPENDIX F OF THE Ada STANDARD

4.10.8. Honeywell GVSC BIF's

The Honeywell GVSC (Generic VHSIC Spaceborne Computer) version of the 1750A contains some Built-In Functions (BIF's) that are made available in the Machine_Code package. These instructions should be used only if the target is the GVSC 1750A. The mnemonics for these instructions are: SIN, ESIN, COS, ECOS, DMBIF, DMBIF2, BIT, ATAN, EATA, SQRT, ESQR.

If the x and y-coordinate arguments of the ATAN (or EATA) instruction are not known to be in adjacent register pairs (or triples), the TWO_OPNDS form may be used to specify both coordinates. The first operand is the x-coordinate and the second operand is the y-coordinate. Subject to fixup rules, the compiler will attempt to find adjacent register pairs (or triples) and move the arguments before emitting the instruction.

4.10.9. Inline Expansion

Routines which contain machine code inserts may be inline expanded into the bodies of other routines. This may happen under programmer control through the use of pragma INLINE, or at optimization levels standard and time (UNIX: -Op2 and -Op3; VMS: /optimize=standard and /optimize=time) when the compiler selects the inline optimization as an appropriate action for the given situation. The compiler will treat the machine code insert as though it were a call. For example, volatile registers will be saved and restored around the inline expansion, and similar optimizing steps will be taken.

4.10.10. Move Macro Instructions

The 1750A instruction set contains no single all-purpose move instruction which will move objects regardless of location (i.e., memory or register) or address mode. Each of the 1750A instructions that does a load or store defines a very specific kind of move with restrictions on the source/destination locations. Unfortunately, when constructing data moves using package Machine_Code, it is impossible to predict if an Ada object will be in memory or in a register, especially in the presence of inlining. For this reason, three ``macro'' instructions are supplied:

Name	Meaning
MOV16	Move 16 bits from the second operand to the first, emitting some combination of the appropriate loads and stores to do so.
MOV32	Move 32 bits from the second operand to the first, emitting some combination of the appropriate loads and stores to do so.

APPENDIX F OF THE Ada STANDARD

MOV48	Move 48 bits from the second operand to the first, emitting some combination of the appropriate loads and stores to do so.
-------	----------------------------------------------------------------------------------------------------------------------------

For example, suppose an integer is assigned to the variable VAR and you want to move the value into register R4. The assembly code insert

```
TWO_OPNDS' (MOV16, (R_AM, R4), (Symbolic_Value, VALUE'Address));
```

may generate one of:

```
LR R4, RX      -- if VAR is currently in a register, RX
L  R4, VALUE   -- if VAR is global and currently in memory
L  R4, xxx, R14 -- if VAR is on the stack at R14+xxx
```

The instruction(s) generated will depend on the scope and binding of VAR. Due to the unpredictability of the location of the source operand, compiling in the Me or fixup=none mode will not cause an error or warning to be emitted if more than one instruction is generated.

4.10.11. Unsafe Assumptions

Please follow the guidelines listed below when writing machine code inserts. Violation of these assumptions may result in the generation of code which does not assemble or code that produces unexpected results.

- Registers which are not volatile across calls (any register other than R0, R1, R2, or R3) must be explicitly saved and restored. Do not assume that a machine code insert routine has its own set of local registers, for if the routine is inline expanded into another routine the local registers might not have been saved.

Values should not be assigned to the frame pointer register in the middle of a machine code insert routine, even if your code saves and restores the contents of the register. A dangerous situation can arise if an exception is propagated through the procedure frame, or if a machine code insert references a variable that uses a frame pointer in the address formula.

- Use an Ada separate body for the routine and make sure that there is no pragma INLINE for it if you wish to guarantee that a routine will never be inline expanded.
- Do not attempt to move multiple Ada objects with a single instruction such as MOV. Although the objects may be contiguous under the current circumstances, there is no guarantee that later changes will permit them to remain contiguous. If the objects are parameters, it is virtually certain that they will not be contiguous if the routine is inline expanded into the body of another routine. In the case of locals, globals, and own variables, the compiler does not guarantee that objects which are declared textually ``next'' to each other will be contiguous in

APPENDIX F OF THE Ada STANDARD

memory. If the source code is changed such that it declares additional objects, this may change the storage allocation so that objects which were previously adjacent are no longer adjacent.

- The compiler will not automatically generate call site code for you if you emit a call instruction, if you emit a call, you are also expected to emit the linkage conventions of the routine you are calling. If the routine you call has out parameters, a large function return result, or an unconstrained result, it is the responsibility of the programmer to emit the necessary instructions to deal with these constructs in a manner that is consistent with the calling conventions of the compiler.
- The 'Address attribute will not always produce an operable address. Whether the attribute is used as an address or a value will be determined by the address mode chosen (Symbolic_Address or Symbolic_Value).
- The compiler will not prevent you from writing register R1, which is used to hold the address of the current exception handler. This provides you the opportunity to make a custom exception handler. However, be aware that there may be considerable danger in doing so. Details of the exception handling convention can be found in the Runtime Modification Manual.

4.10.12. Limitations

The current implementation of the compiler is unable to fully support automatic correction of certain kinds of operands. In particular, the compiler assumes that the size of a data object is the same as the number of bits which is operated on by the instruction chosen in the machine code insert.

Note that the use of X'Address in a machine code insert does not guarantee that X will be bound to memory. This is a result of the use of 'Address to provide a ``typeless'' method for naming Ada objects in machine code inserts. For example, it is legal to say (Symbolic_Value, X'Address) in an insert even when X is found in a register.

4.11. INLINE GUIDELINES

The following discussion on inlining is based on the next two examples. From these sample programs, general rules, procedures, and cautions are illustrated.

Consider a package that contains a subprogram that is to be inlined.

```
package In_Pack is
  procedure I_Will_Be_Inlined;
  pragma INLINE(I_Will_Be_Inlined);
end In_Pack;
```

Consider a procedure that calls the inlined subprogram in the previous package.

APPENDIX F OF THE Ada STANDARD

```
with In_Pack; use In_Pack;
procedure Uses_Inlined_Subp is
begin
    I_Will_Be_Inlined;
end Uses_Inlined_Subp;
```

After the package specification for `In_Pack` has been compiled, it is possible to compile the unit `Uses_Inlined_Subp` that calls the subprogram `I_Will_Be_Inlined`. However, because the body of the subprogram is not yet available, the generated code will not contain an inlined version of the subprogram. The generated code will use an out of line call to `I_Will_Be_Inlined`. The compiler will issue warning message number 2429 to inform you that the call was not inlined when `Uses_Inlined_Subp` was compiled.

If `In_Pack` is used across libraries, it can be exported from the root library after you have compiled the package specification. Note that if only the specification is exported, there will be no inlined calls to `In_Pack` in all units within libraries that import `In_Pack`. If only the specification is exported, all calls that appear in other libraries will be out of line calls. The compiler will issue warning message number 2429 to indicate that the call was not inlined.

There is no warning at link-time that subprograms have not been inlined.

If the body for package `In_Pack` has been compiled before the call to `I_Will_Be_Inlined` is compiled, the compiler will inline the subprogram. In the example above, if the body of `IN_PACK` has been compiled before `Uses_Inlined_Subp`, the call will be inlined when `Uses_Inlined_Subp` is compiled.

Having an inlined call to a subprogram makes a unit dependent on the unit that contains the body of the subprogram. In the example, once `Uses_Inlined_Subp` has been compiled with an inlined call to `I_Will_Be_Inlined`, the unit `Uses_Inlined_Subp` will have a dependency on the package body `In_Pack`. Thus, if the body for package body `In_Pack` is recompiled, `Uses_Inlined_Subp` will become obsolete, and must be recompiled before it can be linked.

It is possible to export the body for a library unit. If the body for package `In_Pack` is marked as exported in the root library using the Ada librarian subcommand `export compilation unit`, other libraries that import package `In_Pack` will be able to compile inlined calls across library units.

At optimization levels lower than the default, the compiler will not inline calls, even when `pragma INLINE` has been used and the body of the subprogram is in the library prior to the unit that makes the call. Lower optimization levels avoid any changes in the flow of the code that cause movement of code sequences, as happens in a `pragma INLINE`. When compiling at a low optimization level, you will not be warned that inlining is not being performed.